Homework 4: Memory and Caching
Due Nov. 21st 2022, 23:59

25 points

1 Task 1: Theory (10 points)

a: What are the requirements for a component to qualify as a memory cache? How do we measure cache performance and what is the expected benefit from using a cache? (2 points)

b: A Direct Mapped Memory Cache is implemented on a 16-bit architecture. The chache can hold 8 blocks of memory, each of which is 32 bytes long. How many bits are necessary in order to represent the tag of each of these blocks? Explain how the cache works and how powers of two are used to parse a byte address and find the correct data stored in the cache. (4 points)

c: A 32-bit architecture with 256 MB of RAM uses demand paging. The operating system blocks 128 MB of memory for privilleged use and distributes the remaining memory to a page table of 1024 entries and a corresponding set of 1024 frames. What is the physical memory size of each page in this system? Explain how the MMU will perform address translation. Explain what a Translation Loookaside buffer is and how it can help speed-up translation. (4 points)

2 Task 2: Array Storage (15 points + 5 bonus)

In this task you will implement storage and fetching into a 2D byte array. This is meant as a proxy to emulate a memory array, which are often relevant when talking about DMMC caches or just plain word alignment. The point of this exercise is to use a 1D character buffer in order to emulate 2D memory arrays in C.

You will declare two functions: `two_d_store` and `two_d_fetch` to implement storage and access from a 2D array of arbitrary data types. We will achieve this result in stages. Follow these steps:

- Implement a function `char* two_d_alloc(size_t N, size_t M, size_t sz)` that allocates a `char*` to store an array of $N \times M$ elements of a given size`sz`. The function takes as arguments the number of rows N, number of columns M, and size sz (in bytes) of each element. Example: `char *buf = two_d_alloc(3,3,sizeof(int))` allocates a 3×3 array of integers for a total of 36 bytes.

- Implement a function `void two_d_dealloc(char *array)` which deallocates a two dimensional array stored in a character buffer.

- Implement a function `int two_d_store_int(int arg, char* array, size_t i, size_t j, size_t M, size_t N)` which stores an integer argument `arg` into a particular row `i` and column `j` of the array. The function should in addition take as arguments a pointer to the array (`char* array`), as well as the numbers of rows `N` and columns `M` in the array. Assume row-major storage order. Make sure your function checks that the arguments are sane (e.g., that the pointer is not `NULL` or that the indeces are not out of bounds) and returns -1 on error or 0 otherwise.

- Implement a function `int two_d_fetch_int(char* array, size_t i, size_t j, size_t M, size_t N)` which returns the value of an integer stored in the memory array. The function should take as arguments the row and column at which the integer is stored, the numbers of rows and columns in the array, the address of the array in memory. Assume the array is initialized in row-major form. Make sure your function checks that the arguments are sane and print an error message to screen if they are not.

- Test the functions you just wrote by allocating a 2D array, storing and then fetching a number of integer arguments. Verify that your implementation works correctly.

- Test your program for boundary conditions.

- Now we will implement two general fetch and store functions that can store any data type of a given size in the array. Define the two functions:

```
int two_d_store(void* arg, char* array, size_t i, size_t j,
        size_t M, size_t N, size_t sz);
void* two_d_fetch(char* array, size_t i, size_t j,
        size_t M, size_t N, size_t sz);
```

  The arguments are defined as in the integer fetch/store functions, except that now we take in a pointer to an element of unknown type `void* arg`, and similarly return a pointer to an element of undefined type as output of the fetch function. The size of elements is however known and passed as argument to both functions (`size_t sz`). You may want to use the `memcpy` function to copy bytes into the array. Note: the fetch function should not allocate new memory.

- Test your implementation. Try storing and fetching into a 2D array of floats.

- **Bonus 5 points:** Demonstrate how memory access and storage works by allocating, storing and fetching into a 2D array containing elements of type `struct Node` defined as:

```
typedef struct Node {
  int payload;
  struct Node* next;
} node;
```

To get the full bonus points, write a helper function `void print_array(char* array, size_t M, size_t N, size_t sz)` that generates a hex dump of the array, printing every element as byte strings. Your print function should start every line by printing the address of the pointer to the first element, followed by a tab character, followed by white space separated hexadecimal byte printout of every element. Use this function to demonstrate what happens when you store node instances in the array. Initialize the `next` pointer of every element you insert to point to the location in the array where you stored the previous element.